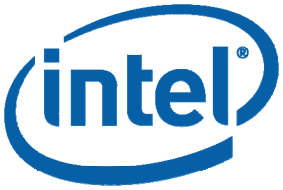


Intel(R) Thread Profiler for Windows*

Getting Started Guide

Intel® Thread Profiler helps you improve the performance of applications threaded with Windows* API, OpenMP*, or POSIX* threads (Pthreads*). Use Thread Profiler to:

- Identify bottlenecks that limit the parallel performance of your multi-threaded application.
- Locate synchronization delays, stalled threads, excessive blocking time, and ineffective utilization of processors.
- Find the best sections of code to optimize for sequential performance and for threaded performance.
- Compare scalability across different numbers of processors or using different threading methods.

Overview

This guide presents a threaded code example and shows you how to use Thread Profiler to identify performance issues. After completing this guide, you will be ready to analyze and optimize your own code using Thread Profiler.

Contents

Disclaimer and Legal Information

Revision History

Document Number	Revision Number	Description	Revision Date
	1.0	Initial release.	2003
313032 US	2.0	1. Applied new template. 2.Updated content for 3.0 product.	June 2006

§

Document Number: 313032 US

1 Build the Sample Code

The `Primes` example is a prime number generator that uses the Windows* threading API. The code identifies and tallies the prime numbers from one to 100,000 by testing whether odd numbers are evenly divisible by smaller odd factors. The code generates four threads to do the work using Windows* API.

1. Open the `Primes.dsw` Project Workspace file in Microsoft* Visual Studio. By default, this project is installed with Thread Profiler in:
   `C:\Program Files\Intel\VTune\tprofile\samples\Primes`.
2. Build the `Primes.dsw` project.
   This project includes the following options: `/Zi` to include symbols, `/Od` to disable optimization, `/fixed:no` linker option to make code relocatable, and `/MDd` or `/MTd` to build with thread-safe run-time libraries. These options are required to enable Thread Profiler to provide you with detailed information, such as variable names and line numbers associated with performance problems.
   The project creates a debug image, `Primes.exe`.

TIP: To quickly start using Thread Profiler, print this short guide and walk through the example provided.

TIP: See Thread Profiler online **Help** for more hints on options and building applications.

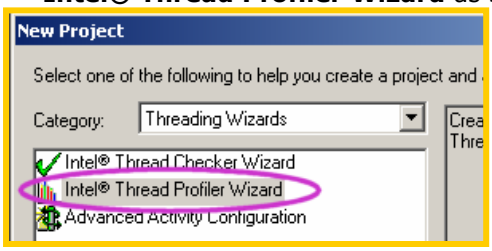
NOTE: Your applications do not need to be rebuilt with these options in order to benefit from Thread Profiler analysis. However, using the recommended options helps Thread Profiler give you the most detailed information about your code.

NOTE: Thread Profiler's analysis of programs that use OpenMP* compiled using supported Intel compilers differs from the examples presented in this guide.

2 Collect Data

The **Intel® Thread Profiler Wizard** enables you to easily generate and collect performance data on your threaded code. To use the wizard:

1. Start the Intel(R) Thread Profiler from, for example, **Start** > **All Programs** > **Intel(R) Software Development Tools** > **Intel(R) Thread Profiler** > **Intel(R) Thread Profiler.**
   Or, double-click the Intel(R) Thread Profiler icon on your desktop.
2. In the **Easy Start** dialog box or from the main toolbar, click the New Project button, to open the **New Project** dialog box.
3. In the **Category** drop-down box, choose **Threading Wizards** and select **Intel® Thread Profiler Wizard** as shown:



4. Enter a **Project Name**, for example, *PrimesProject*.
5. Click **OK**. The **Intel® Thread Profiler Wizard** opens.
6. Under **Threading Type**, select **Threaded (Windows* threads, POSIX* threads, and OpenMP* analysis)**.
7. Under **Launch an application**, click **[...]** to navigate to the debug build of `Primes.exe` you built, located in…`\Primes\Debug\Primes.exe`.
8. Click **Finish** to complete the wizard.
   Thread Profiler instruments your application, executes it, collects and displays data results.

The exact results vary depending on your system configuration, but you should see two color charts, one with horizontal bars, one with vertical bars. Congratulations! You are now ready to identify and locate bottlenecks that are limiting the parallel performance of the sample software.

TIP: If you do not see results, open **Help** > **Search** and search "**Troubleshooting Thread Profiler"** for possible causes and solutions.

3 Analyze Results and Correct the Code

In this section, you will walk through Intel® Thread Profiler's main views to identify performance issues related to threading. You will then consider ways to improve the performance of the sample code.

3.1 Profile View

By default, Thread Profiler displays **Profile** and **Timeline** views as shown in Figure 1:

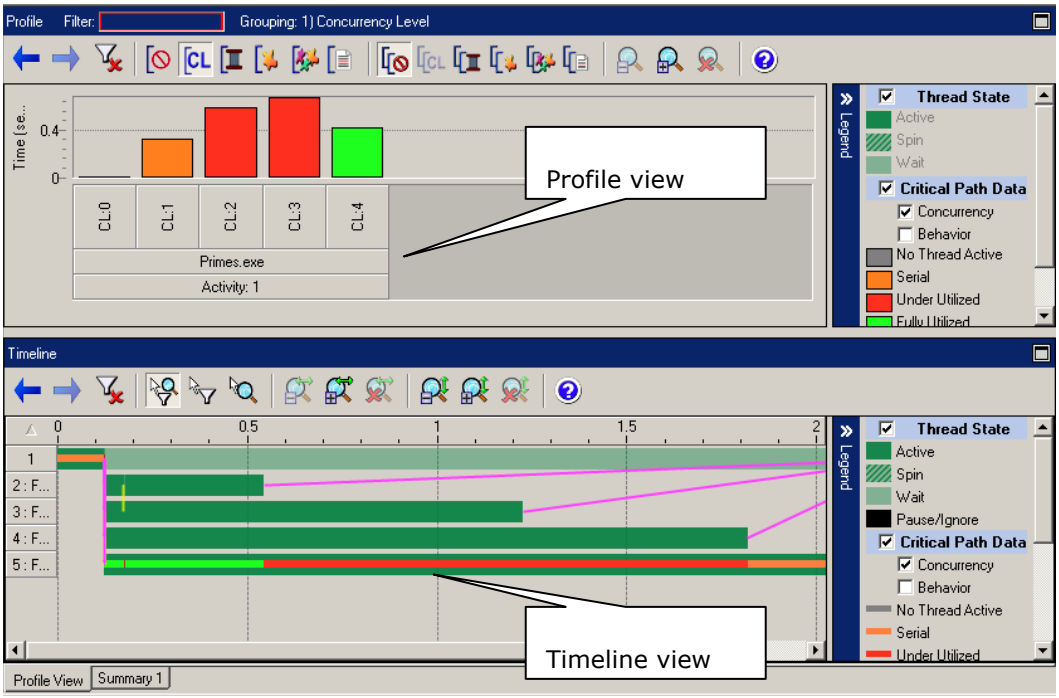


Figure 1 Profile and Timeline views are the default views for Intel(R) Thread Profiler Activity results.

The **Profile** view (on top) displays a high-level summary of the time spent on the critical path, decomposed into time categories. The **Timeline** view (on bottom) illustrates the behavior of your program over time.

By default, **Profile** view initially shows results grouped by **Concurrency Level**, the number of active threads executing at the same time on the critical path. It includes

threads which are currently running or are run queued and not waiting at a defined waiting or blocking API.

You can also group data results by **Objects** to compare time impact due to different software objects. Double-click one of the bars in **Profile** view to group by **Objects**.

Now the **Objects** button,  on the **Profile** view's toolbar is selected as shown in Figure 2:

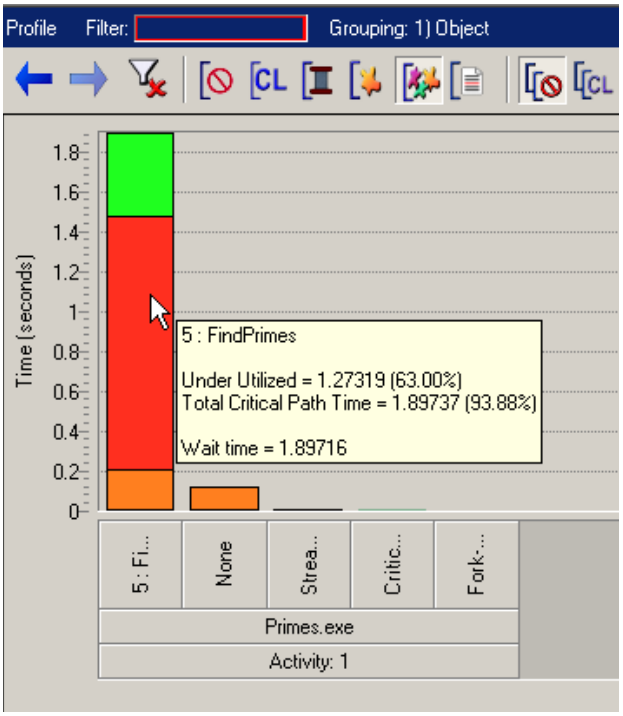


Figure 2 Profile view grouped by Objects. In this case, the majority of time was spent in Under Utilized (red) time.

Hover your mouse over a column to see more detailed data. For example, in Figure 2, the first column corresponding to `5:FindPrimes` shows a majority of time was **Under Utilized** (red) with portions of **Fully Utilized** time (green) and **Serial** time (orange).

Thread Profiler shows columns corresponding to objects that cause contention on the critical path. You can see more detail about how time was spent on the Critical Path by checking **Behavior** in the **Legend** as shown in Figure 3:

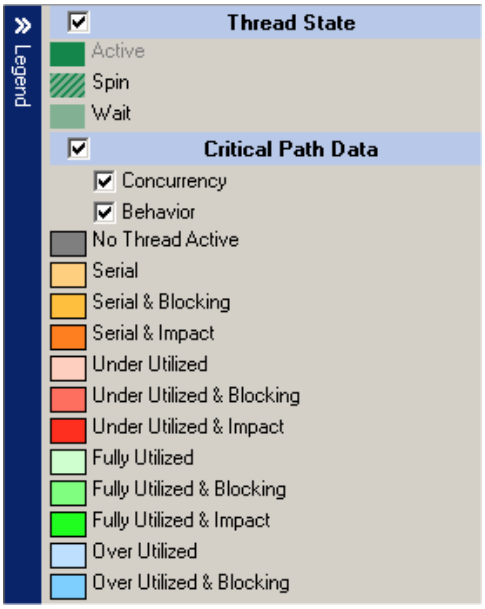


Figure 3 Profile view's Legend enables you to select different levels of information to display by checking or unchecking options such as Concurrency and Behavior.

In this example, most of the time on the critical path is attributed to a **Fork-Join** object associated with thread 5`: FindPrimes`. The thread name is based on the function name of its entry point. The fork-join object implies that there could be imbalance between thread 2 and the thread waiting for thread 2.

In **Profile** view, you can also:

- Click any bar to display the detailed timing information in the **Statistics** table under the graph.
- Double-click any bar to "drill down" to the next logical level of grouping.
- Right-click any bar and select **Filter Selection** to filter by the current selection.
- Right-click a bar and select **Filter and Group by >**to filter and group by **Concurrency Level, Thread, Object, Object Type,** or **Source Stack**. You can also use the corresponding toolbar buttons to group data by the same categories.
- Apply second-level grouping using the secondary set of toolbar buttons:

- For certain types of data, you can right-click and select to **Filter and Show Source Locations**.

3.2 Timeline View

Timeline view shows the contribution of each thread to the total program, whether on the default critical path or not.

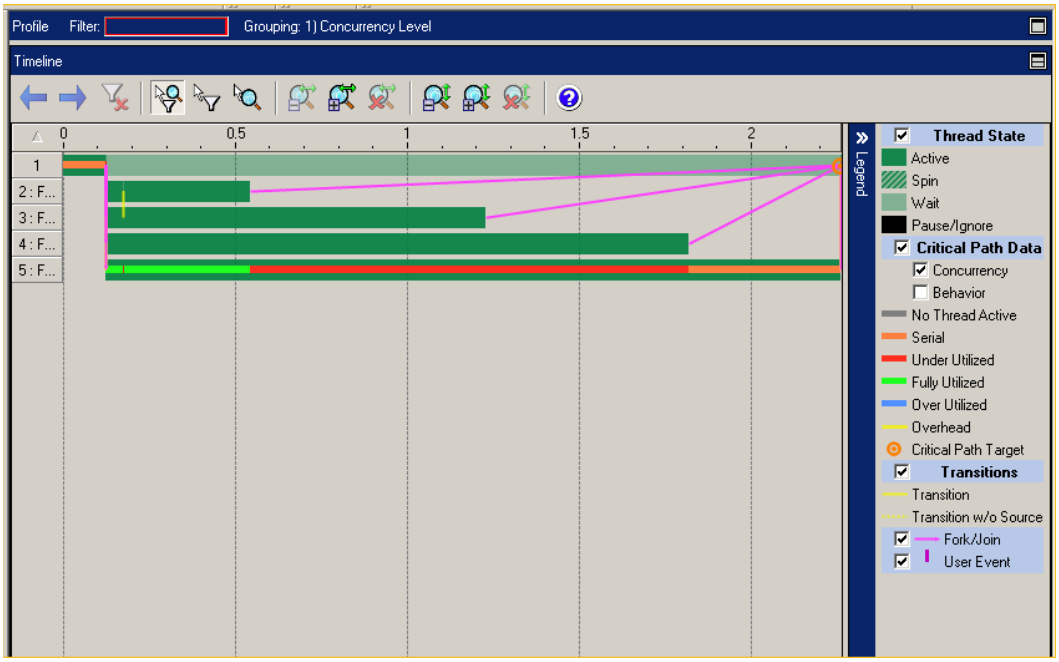


Figure 4 Timeline view shows the behavior of a program over time and across threads.

Notice in Figure 4 that the four created threads take increasingly longer to run. Thread 2, represented by the next-to top bar, works on a set of small numbers that take little time to check. Thread 3 checks a set of larger numbers, taking longer. Threads 4 and 5 check larger numbers, taking longer to complete. Though the program accomplishes its task, the loading is clearly unbalanced. In the next section, you will explore ways to improve the parallelism in this code example. Meanwhile, take a closer look at the **Timeline** view.

Thread Profiler tracks the flow of all threads in the application. The longest continuous path through the flows is defined as the *default critical path*. You can also set your own critical path targets to focus your analysis on a particular path or thread by right-clicking and selecting **Change** or **Add Critical Path Target** where you want the new target to appear. Use this feature to focus on a thread that is of particular interest to you, even if it is not on the default critical path.

In **Timeline** view, you can use the **Legend** to hide or show the corresponding time categories that appear in the **Profile** view. You can also chose to hide or show

Transitions, **Forks** and **Joins**, and **User** events that you defined. When you mouse over the different elements in the **Timeline** view's **Legend**, the corresponding elements flash in the graph, helping you to locate or identify different elements in the graph.

In Timeline view you can also:

- Drag your mouse over a section of the **Timeline** view and release to zoom in on that section.
- Use the horizontal and vertical zoom buttons on the toolbar,  to help you focus on a particular section of the graph. This feature is particularly useful when you have vast quantities of densely displayed data.
- Double-click a bar to drill-down to **Source** view to see where an event is happening in your code.

3.3 Time Categories

Thread Profiler breaks time into different **Time Categories**, represented in the graphs and **Legends** by different colors. In this example, the critical path is composed of the following time categories, with **Behavior** checked:

- **Serial & Impact** time (orange) indicates serial portions of the code.
- **Under Utilized & Impact** time (red) indicates that the code is not fully utilizing all processors.
- **Fully Utilized &Impact** time (green) indicates that portions of the code demonstrate good processor utilization.

Ideally, on a multi-processor system, your code should be characterized by a predominance of **Fully Utilized** time (the sum of all the greens). All other time categories indicate opportunities for improving performance. In this case, more efficient code would be characterized by more **Fully parallel time** than **Under Utilized time**, so there are opportunities to enhance performance.

To understand time categories:

1. In the **Profile** or **Timeline** view, press **F1** to open the online **Help**. A related help topic opens, as shown in Figure 5.
2. Click the link **Legend for Profile and Timeline Views** to open a topic that describes the elements of the legend.
3. Read the **Help** topic and follow links to related topics.

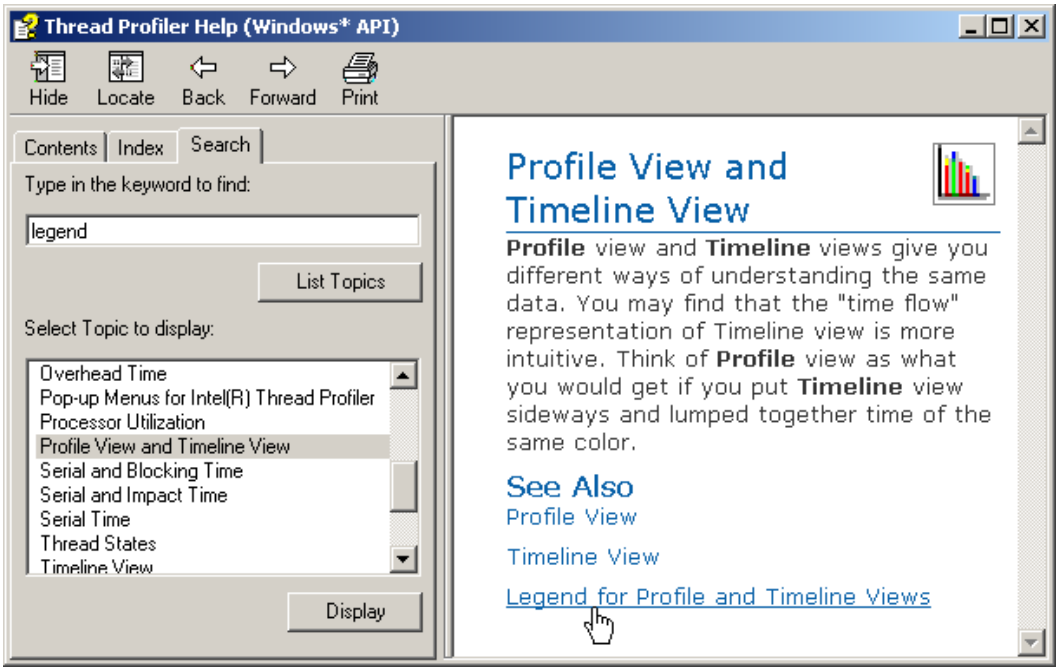


Figure 5 Help topic for Profile and Timeline View. Press F1 to access help topics.

In the **Help** you can also:

- Click entries in the **Contents** pane to jump to a topic.
- Click the **Search** tab to open the search window to find a topic on a particular subject.
- Find tips for improving your code such as **Dealing with Cruise Time** for valuable tips on ways you can increase parallelism during cruise time.
- Click the locate button on the **Help** toolbar, 

to browse to related topics in the **Contents**.

4 Correct the Code

To correct the imbalance and reduce the overhead identified during analysis, try the solutions presented in this section. Corrected versions of the sample code are provided for you to check your results and note the differences.

4.1 Solution: Increase Parallel Execution Time

The revised code in the sample `PrimesBalanced.cpp` interleaves the numbers each thread checks so that each thread works on both small and large values. To verify that the imbalance problem identified by Thread Profiler as a performance issue is indeed fixed, do the following:

1. Build the modified code which is in `PrimesBalanced.cpp`.
2. Right-click the project name to create a new Activity. Collect data using **Intel® Thread Profiler**. You can quickly create a new Activity result for the modified example, `PrimesBalanced.exe` by pressing the **F5** key, or by clicking the Run Activity button  .

4.2 Analyze Results

In **Profile** view, you should see improvements in the form of longer **Fully Utilized** (green) execution times. Note the little bit of **Overhead** (yellow) which indicates the delay of a transition of the critical path from one thread to the next.

In **Timeline** view, the threads now show improved workload balance over the original example.

4.3 Eliminate Overhead

You can further improve performance by eliminating overhead associated with entering and leaving the `CRITICAL_SECTION`**.** Since you really only need to control access to the incrementing of variable `PrimeCount`, you can use the `InterlockedIncrement` API to reduce overhead. See the `PrimesLockfree.cpp` sample for an implementation of this fix.

5 Next Steps

Before using Intel® Thread Profiler to improve performance, use the **Intel® Thread Checker** to verify that your code is free of conditions that could lead to inconsistent results. Find details about Thread Checker and other Intel software development products at: http://www.intel.com/software/products/.

To get the most out of Thread Profiler, explore the following resources:

- **Online Help** is the product's complete user's guide. Use **Help** to learn about features not mentioned in this Guide including remote data collection and other advanced options.
  Open **Help** by pressing the **F1** key.
- **Samples** provide additional code examples for you to explore. Use them to learn to identify and resolve other types of threading errors.
  Open code samples in the `tprofile\samples` folder. Read the associated ***Guide to Sample Code***, `CodeExamplesGuide.pdf`, available in the `tprofile\Doc` folder.
- **Release Notes** include key product details. See the Release Notes for updated information on requirements, technical support, and known limitations.
  Open **Release Notes** from, for example, **Start > Programs > Intel(R) Software Development Tools > Intel(R) Thread Profiler > Intel(R) Thread Profiler Release Notes**.